Numerical data visualization Documentation

Son Nguyen
906900
BSc. Data Science

May 2022

1 General description

The program can read data from a JSON file defined by user and create 3 types of charts: line chart, bar chart and pie chart. The line chart has lines with given start and end points, each line is drawn with a different color and there is a legend showing which color corresponds to which set of points. The user has options to name the axes and determine the size of a grid.

The bar chart shows columns with heights from given data. Under each column is the name of the data. Users have the option to name the vertical axis.

The pie chart shows a circle with different segments in different colors representing the proportion of the data. There is also a legend showing which color, name and numerical data of the data. The total sum of the data make the full circle.

2 User interface

The first step is to prepare a JSON data file. For making a line chart the data file needs to have this structure:

```
{
    "data": [
        [{"x": -0.5, "y": 1.5}, {"x": 5, "y": 5}],
        [{"x": 96, "y": 37}, {"x": -67, "y": -3}]
    ],
    "config": {
        "name": "line chart",
        "grid": "10",
        "y": "y",
        "x": "x"
    }
}
```

The ”data” field is required while the ”config” field is optional. The data is a list of pair of points, each point must have both values ”x” and ”y” followed by a real number. Please be aware that there

must be **NO** trailing comma behind the last member of a list (e.g. last point in the pair, or last pair in the "data", or after "config").

The available options in config are:

- "name" for chart name, value is a string
- "grid" for size of grid, value is a string but only a positive number inside will create a grid
- "x" for label for x axis, value is a string
- "y" for label for y axis, value is a string.

Bar chart file:

```
{
    "data": {
        "A":10,
        "B":20
    },
    "config": {
        "name": "bar chart",
        "y": "y"
    }
}
```

Pie chart file:

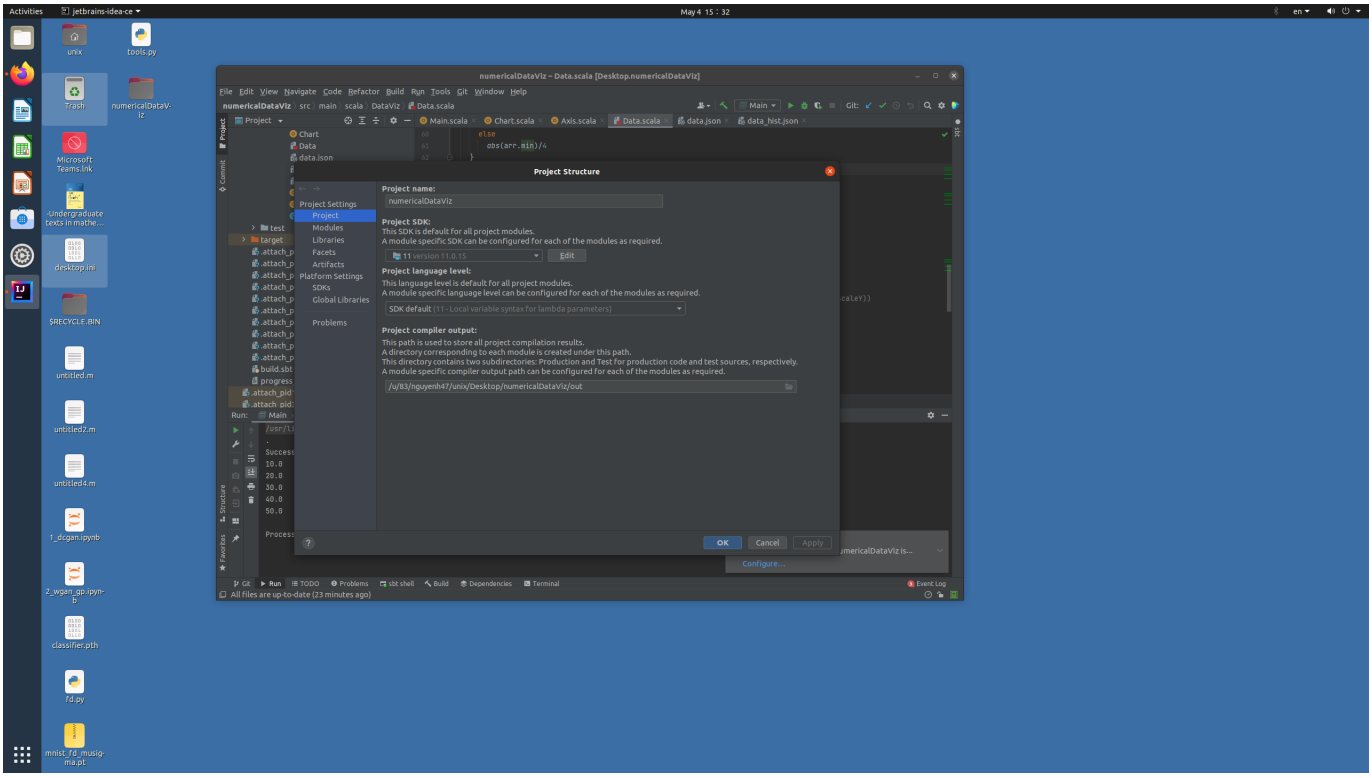
```
{
    "pieData": {
        "A":10,
        "B":20
    },
    "config": {
        "name": "pie chart"
    }
}
```

In the data file, the variable names must be kept like as shown here. Notice in the file for pie chart the variable for data is named **pieData** instead of data as usual. This is because PieData and HistogramData has similar type of data so I give them different names to differentiate. Also one more time **NO** trailing comma behind the last member of anything.

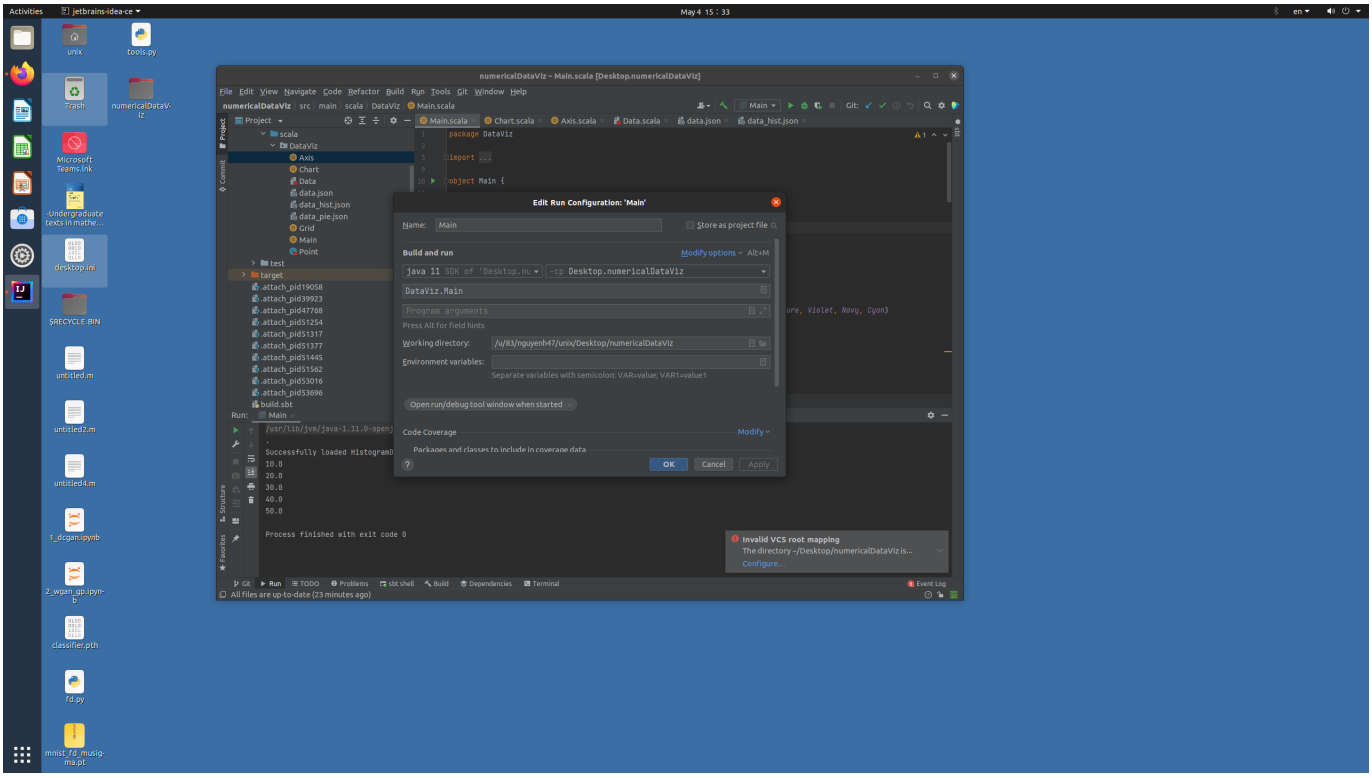
After the data file is ready, put the full location of the data file in file Main.scala as variable filename. There are also predefined JSON file under the names data.json, data_hist.json and data_pie.json that are in the correct structure and can be modified to use. This is not an elegant way to select a file and if I had more time I would definitely improve this.

I tried running the program at one of Aalto's Linux computers and it worked so hopefully it is the same for your computer:

First open using Intellij, then download Scala plugin. Then in project structure choose SDK version 11 like in the picture.

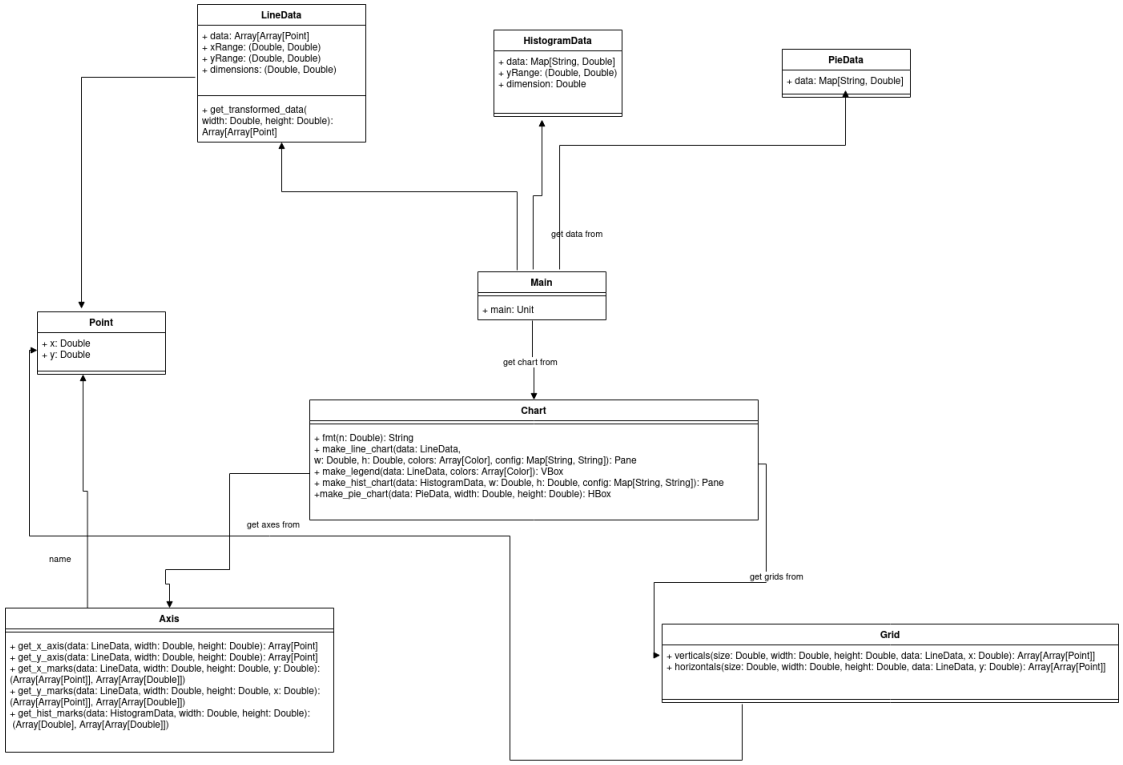


Then when press run in Main.scala there might be a pop up about run configuration. Try selecting like in the picture.



3 Program structure

Below is the UML of the program. The structure is quite simple, the main class gets data from the Data classes then use those data to call methods from Chart, Axis and Grid to create charts.



4 Algorithms

4.1 Line Chart

4.1.1 Intervals

We start with processing our input data. There are xRange and yRange variables that are the intervals that have all the data inside. For example, if xRange $= (1, 2)$ then all data points have their x-coordinates $1 < x < 2$. These intervals are calculated by first locating the maximum and minimum coordinates, then add some "padding" around the interval: (min - unit, max + unit) where

$$\text{unit} = \text{pow}(10, \text{floor}(\log10(\text{max - min}))) \ / \ 4$$

if max $\neq$ min.

In case max = min and non-zero (when plotting a single point or a horizontal/vertical line)

$$\text{unit} = \text{min} \ / \ 4.$$

If max = min = 0 then the padding unit is 10.

The padding is added because it would help plotting looks nicer. In my plotting algorithm (below), without the padding the points that have the max/min coordinate will be at the edge of the plot, so

the padding helps those points lie completely inside the plot. It also helps when the interval is 0 as that interferes with my other algorithms. In essence, the calculation above makes the padding have reasonable size compared to the interval so that we won't have a small interval with huge paddings around or large interval with too insignificant padding. If the interval = 0 (max = min) then we add 1/4 of the value around the point, and in the other case the rule is:

- $1 \leq$ interval $<10 \rightarrow$ unit = 1 / 4
- $10 \leq$ interval $<100 \rightarrow$ unit = 10/4
- $100 \leq$ interval $<1000 \rightarrow$ unit = 100/4
- $\cdots$

4.1.2 Transform data for plotting

We need to transform data points for plotting because usually a graphics program will (0, 0) at the top left corner and larger x goes to the right and larger y goes to the bottom, so we can't just put raw coordinates in, especially for negative ones.

First I set a fixed width and height for the plot then try to fit everything in that window. An x-coordinate is transformed to be plotted in that window by a function:

$$f(x) = (x - x_1) * w/(x_2 - x_1)$$

in which w is the predetermined width of the plot, (x_1, x_2) is the interval calculated in 4.1.1. Since x_1 is the smallest x-coordinate, we will let it be on the leftmost edge, and similarly x_2 will be on the rightmost edge, we can see that $f(x_1) = 0$ and $f(x_2) = w$. Then $(x - x_1)$ gives the distance from this point to the left edge, but then we need to scale our distance to fit with the fixed width, so we multiply by the ratio $w/(x_2 - x_1)$. At this point we can see that if $x_1 = x_2$ then these computations will have trouble, so some padding to guarantee $x_2 - x_1 > 0$ is needed.

The transformation for y-coordinate is similar:

$$g(y) = (y_2 - y) * h/(y_2 - y_1)$$

with (y_1, y_2) in the intervals and h is the fixed height. It's not $(y - y_1)$ because the way larger y is set to go down in graphics. We can see that $g(y_2) = 0$ will be on top and $g(y_1) = h$ will be on the bottom of the plot This is implemented as function get_transformed_data in class LineData.

4.1.3 Axis

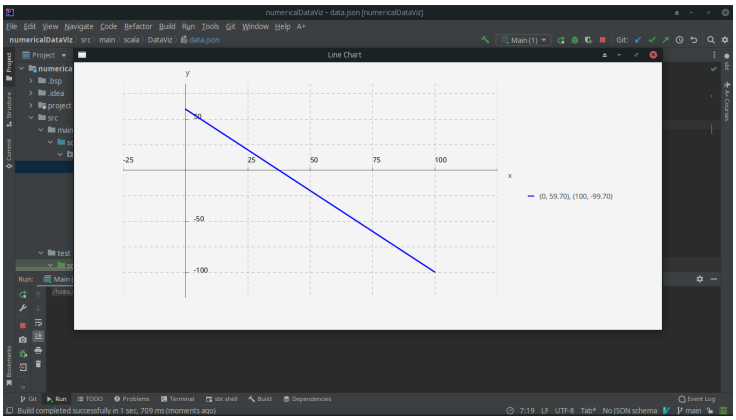
I decided to draw axes with full length, so to draw x-axis I only need to find its y-coordinate and for y-axis its x-coordinate. For x-axis, if we have data points with both positive and negative y-coordinates, then the x-axis will be somewhere inside the plot, but if the points all have positive y-coordinates then the x-axis will lie at the bottom of the plot and vice versa. It is also similar for the y-axis, it will be somewhere inside the plot or left or right edge. When the axes are inside, to find their exact location I also use the data intervals and the transformation function above, using the fact that x-axis has y = 0 and y-axis has x = 0. These are implemented as function get_x_axis and get_y_axis in object Axis.

The algorithm to draw the marks on the axes is a bit more complicated. I will explain the algorithm for x-axis as it is similar for y-axis. Each mark is a short line so I need the location of its two end

points: one is on the axis and the other is a bit upper (or lower if x-axis is at the top of the plot), I let that distance to be 5 pixels. We also need number labels for the marks, which will have the same x-coordinates as the marks, but slightly larger (or smaller) y-coordinates, I let that to be 15 pixels. For spacing between marks (or determine how many marks to draw), I again make use of the unit computation in 4.1.1 for the data intervals. Initially the unit = pow(10, floor(log10(interval))). If the interval length $\leq$ unit * 2 then unit = unit / 4, and if interval $<$ unit * 5 then unit = unit / 2. The spacing between each mark is equal to one unit, therefore for intervals that are not large compared to the unit, I need to make the unit smaller otherwise there will be too few marks. For example if the interval is (0, 100) then the unit is also 100 and there will be only two marks 0 and 100, so by scaling it down to unit = unit / 4 = 25, we now have 3 more marks 25, 50, 75. On the other hand, when interval length $\geq$ unit * 7 then unit = unit * 2 so that there will not be too many marks drawn on each other. Now as we have our unit, we will keep going up until we hit the large end of the interval, each time increment with one unit, then keep going down until we hit the small end of the interval, each time decrease by one unit, then those values will be the x-coordinates of the marks, then we scale those values with $f(x)$, and use y = y-coordinator of the x-axis or y + 5 or y + 15 as explained above as the y-coordinator for the points. The functions are implemented as get_x_marks and get_y_marks in object Axis.

4.1.4 Grid

Grids are a collection of vertical and horizontal lines, so drawing them is similar to x-axis and y-axis. For vertical lines we only need to know their x-coordinates. The spacing between the lines is similar to the markings of the axes, we have the input from the user indicating the grid size which is also the space between lines, so we scale that unit to fit the width and keep drawing until we are out of the plot width. But then I had the problem of determining where to start drawing. Initially I start from the left edge of the plot, but then the grids do not align with anything and do not seem so helpful. Then I thought about aligning them with the markings on the axes, but then the grid size dictated by the user and the space between markings are not always equal so I could not think of a sensible solution for that. Eventually I decided to start from the axis: the first grid lines will be one unit away from the axes. Here is an example with grid size 25:



4.2 Histogram

4.2.1 Data

For histograms, data are processed almost the same as with line charts. The difference is that the lower end of the interval is always 0 and the padding is only added to the upper end of the data interval.

4.2.2 Plotting

The column width for each type of data is calculated by

$$w* = (w - n * s)/n$$

in which $w*$ is the column width, w is the plot width, n is the number of columns and s is the predetermined gap between each column. This calculation fixes the space between column then finds out how much space is left after using $n * s$ for n columns, then divide that by n to get the width of a column. I would prefer a dynamic s to a fixed one but I have not been able to come up with any alternative.

4.3 Pie chart

For pie chart, not much data processing is needed except for checking if there is a negative value since that does not make sense in making a pie chart. In that case the program will halt and print a message to users. The chart is drawn by calculated the proportion of each data over the total, then multiply by 360 to get the pie angle that the data gets. One thing that needs attention is when the total sum is 0 then division by 0 can happen, therefore the sum is checked and if it is 0 then it is changed to 1 to avoid division by zero but still maintain the result since for each data it is $0/1 = 0$ so they get 0 from the pie.

5 Data structures

The most used data structure in this project is Array since I usually need to store a list of coordinates or a list of lines. In the Axis and Grid object where I need to keep adding new line to the Array, mutable Array is needed. I also used a Map to store the configuration from user input.

For the GUI, first the plot is drawn on a Canvas, then the Canvas is put inside a Pane. The Legend is drawn as a list of HBox, which is then put into a VBox. Then the Pane and the VBox is put in a HBox which is the root of the Scene.

6 Files and Internet access

The program takes as input a JSON data file with structured described in part 2.

7 Testing

Due to the graphical nature of the project, it is more straightforward to test the program by manually inspecting the output plot. I have tried running the program with some cases that I could think of:

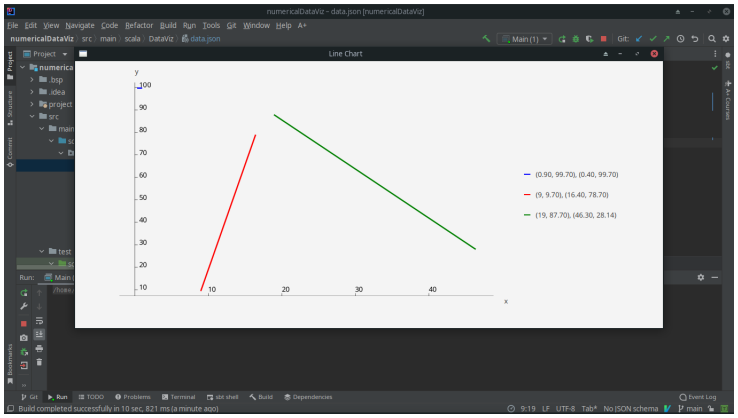


Figure 1: Inputs with all positive coordinates

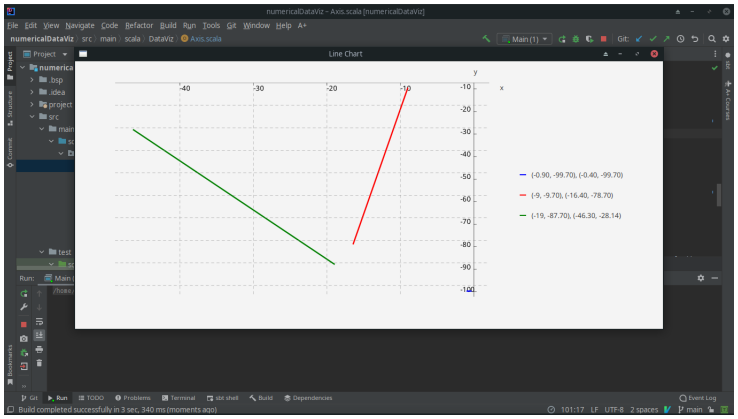


Figure 2: Inputs with all negative coordinates

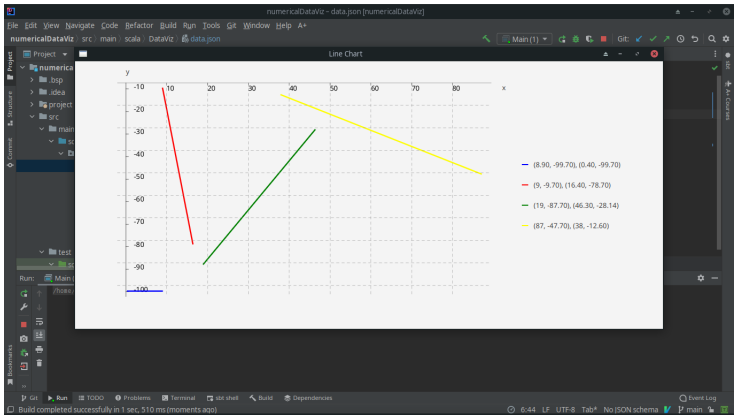


Figure 3: Inputs with all positive x-coordinates and negative y-coordinates

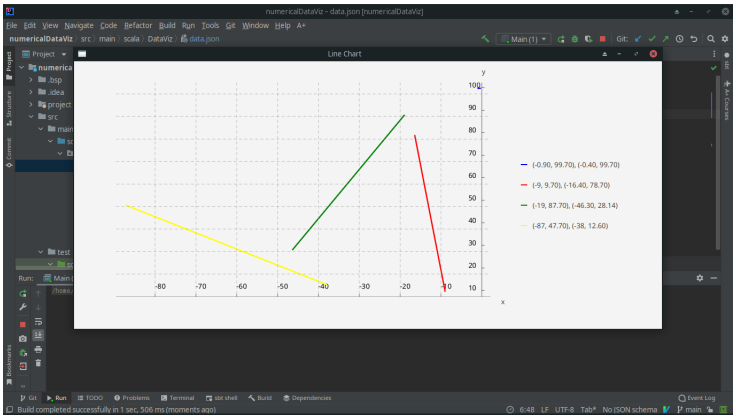


Figure 4: Inputs with all negative x-coordinates and positive y-coordinates

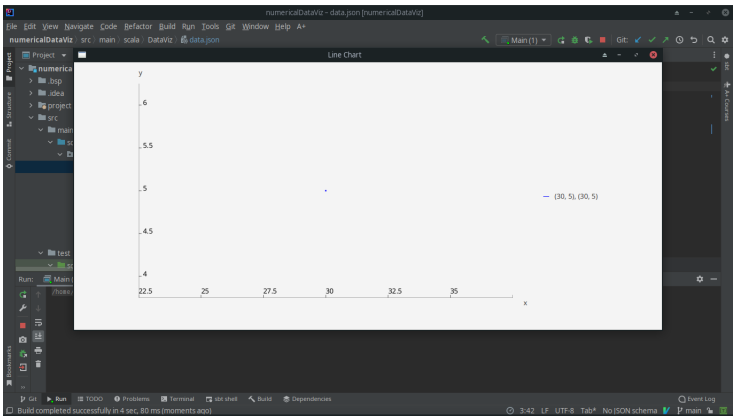


Figure 5: Plotting one point

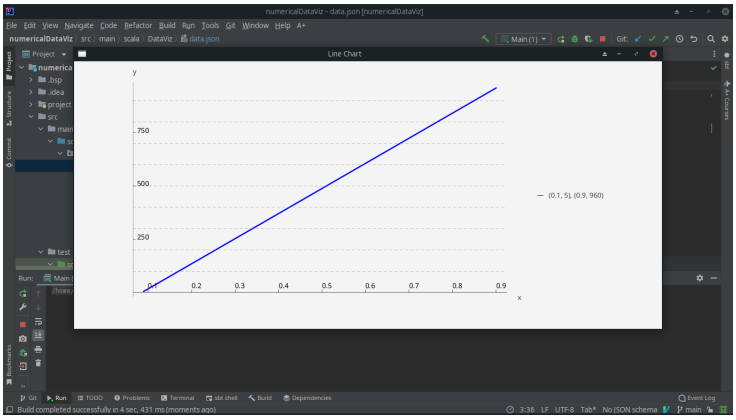
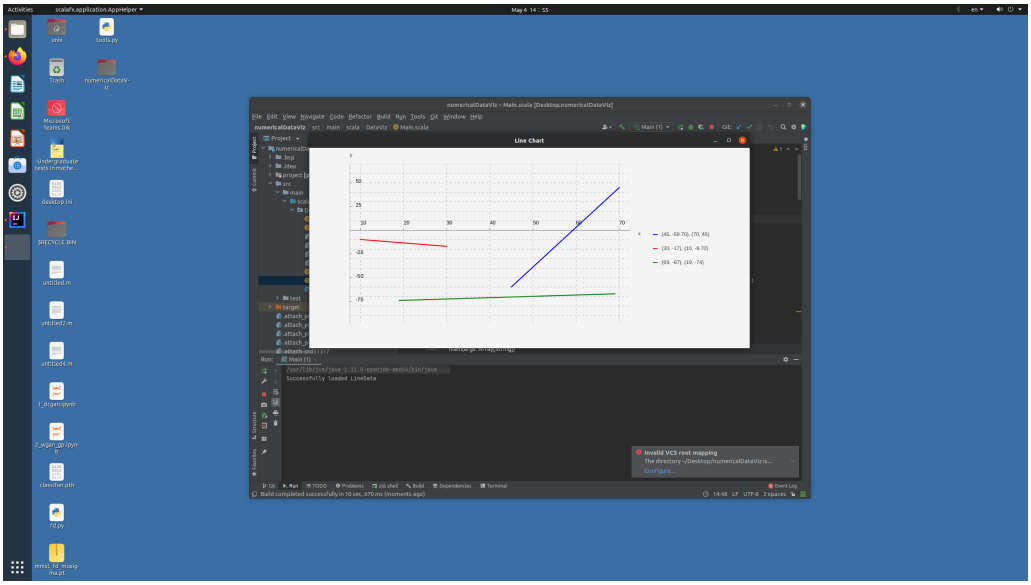
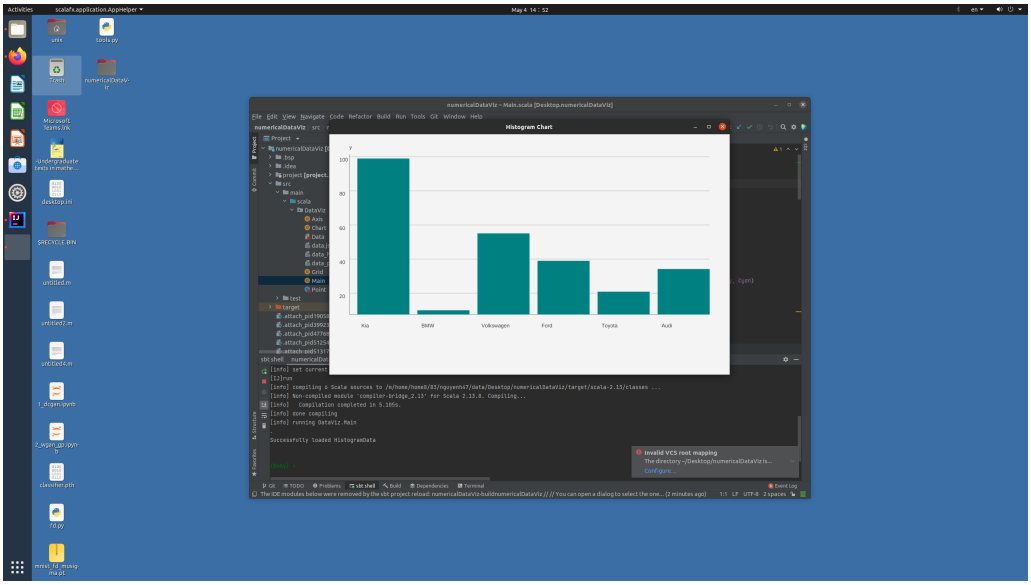


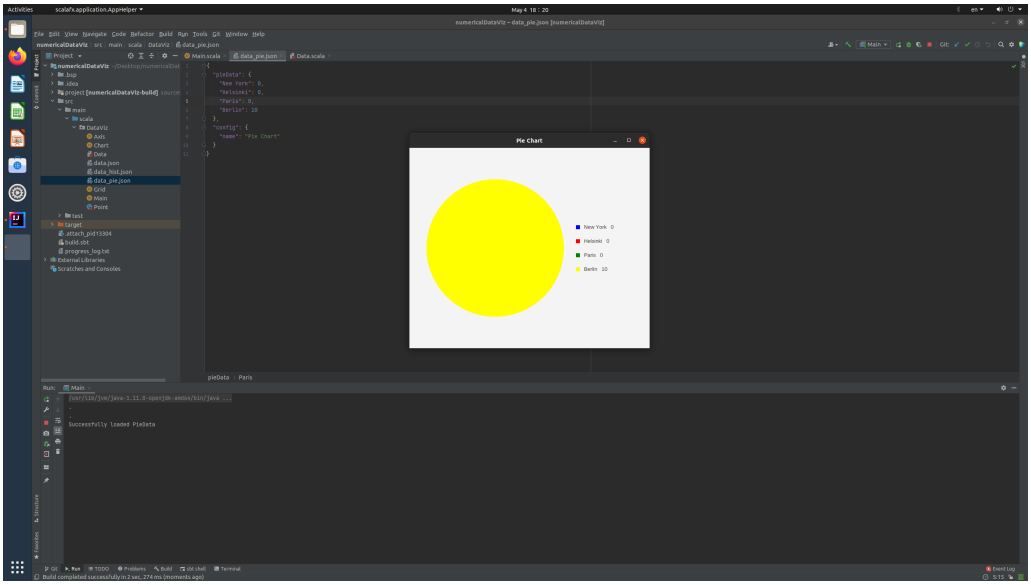
Figure 6: Plotting one line with large difference in values

I also used a unit testing to check the padding feature in LineData class.

Before submitting, I also tried running the program at one of Aalto's Linux computers:


Line Chart


Histogram Chart
Kia
BMW
Volkswagen
Ford
Toyota
Audi


Pie Chart
New York 0
Helsinki 0
Paris 0
Berlin 10

8 Known bugs and missing features

I am not too confident about the display of the labels on the axes and I think for certain inputs there can be numbers drawn on each other. I really tried to alleviate the problem but since the range of input value is too large (anywhere from very small number to very large number), it is difficult to come up with a general solution to handle all inputs nicely. In this project, a lot of calculations has division so division by zero can happen even though I have really tried to identify and get rid of a few. Due to the time constraint I could not test the project thoroughly so I am certain there are many other bugs that I could not find.

One major shortcoming of this project is the amount of code repeated in different places since processing tasks for different data are quite similar. I am aware of the problem, however using inheritance to let the data types share the resources is difficult when combined with Circe to read and decode JSON file, and the time constraint does not allow me to properly research so I had to go with the repetitive design. One missing feature already mentioned in part 2 is that there is not a proper user interface for opening data file. Apart from that, the formatting of the Double type number for the axes and legend is not really satisfying, especially when input data has small number such as 0.0001.

9 3 best sides and 3 weaknesses

3 best sides:

- The program works, at least for quite a bit of input
- The code is quite simple
- Even though I had to rush coding in the last few days, I did not grow to hate programming

3 weaknesses:

- Repetitive design
- The axes labels are not handled really well
- Big clunky methods that need to be divided into smaller ones

10 Deviations from the plan, realized process and schedule

The implementation of this project deviated from the initial plan drastically. Due to taking too many courses to rush for graduation, I was overwhelmed and could not manage time well. As a consequent, I did not make any progress in the project for a long time even until the final check point and had to rush in the last few days. From this I not only learn some programming skills but also the importance of time management and how fast things can go off-track.

11 Final evaluation

There are a lot that I want to improve about the program, but mostly I want to "modulize" it more to get rid of the repetitive design. I also want to have better user interface and better way to input data since editing JSON files seems tedious and there are countless times I forgot the trailing comma, leading to failed reading data. I also learn a lot about time management through this project, mostly through the hard way. I'd like to thank Taige for always says happy coding, which reminds me to enjoy it even under great pressure.

12 References

Lewis, M. C., Lacher, L. L. (2017). Introduction to Programming and Problem-Solving using Scala. 2ed. CRC Press.

ScalaFX. https://www.scalafx.org/

Scala Standard Library. https://www.scala-lang.org/api/current/index.html